ECM3401

Literature Review

A History of Computer Chess

Ben Keen

November 2009

Abstract

This paper provides a look into the rich history of computer chess, ranging from the mid-1940s to present day. It gives a historical perspective of how computer chess has changed over the years, and the ways in which it has remained the same.

The main body of the review is a comparison of the algorithms behind computer chess programs, a description of how they work and the history of their creation. This is further split into three sections which describe the main issues affecting the development of a computer chess program. These are board representation, tree searching, and positional evaluation.

Firstly we look at board representation, and show how the most basic idea of a chess board can be turned into a working board representation. We then look at more recent approaches, and how these compare and contrast.

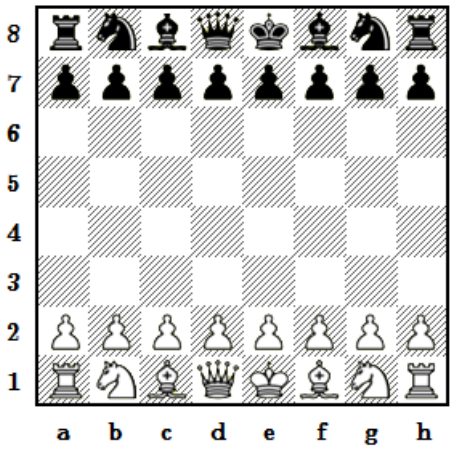
Next is the largest section of the report - the Tree Searching section. This gives a brief introduction into how a computer will go about searching for the right move, and give an overview of the algorithms that have been created to do this.

Lastly is the positional evaluation section, which details how a computer chess program values different board positions.

I certify that all material in this dissertation which is not my own has been identified.

(Ben Keen)

Contents



"Chess is the art which expresses the science of logic"
Mikhail Botvinnik

1 Introduction

Computer Chess has a long and interesting history. The very first machines, such as the Turk [Frey, 1977, p.1-3], that were purportedly capable of playing chess were simple hoaxes and trickery. From these humble beginnings, modern chess computers have evolved to be capable of beating the very best in the World [IBM, 1997]. Getting to that stage, however, has been the work of around 60 years.

Being a perfectly known game, and the traditional dominion of logicians and mathematicians, chess was a fundamentally attractive prospect for academics. The early breakthroughs in computer chess came in the middle of the 20th century, with the work of eminent scientists such as Alan Turing and Claude Shannon. Turing was convinced that games in general were an ideal model system for machine intelligence [Levy and Newborn, 1982, 44-45], an opinion which is very much taken to be true today. During the late 1940s and early 1950s, there came two very important events in the history of computer chess. In 1950, Claude Shannon published *Programming a Computer for Playing Chess*, the first paper on the subject [Shannon, 1950]. At around the same time, Alan Turing, working with D. G. Champernowne, created TUROCHAMP - a move analyser for chess, and the first acknowledged chess computer [Levy and Newborn, 1982, 44-45]. Whilst not actually capable of running on a computer in the modern sense, TUROCHAMP enabled by-hand simulation of the computer's actions.

As time passed, and computers evolved from simple difference engines into true Turing Machines, Chess continued to be an attractive problem for Computer Scientists to tackle.

In order to create a computer program which is capable of playing the game of chess, a number of challenges had to be solved by the early pioneers. These problems are mostly still apparent today, and as such modern chess computer systems are relatively similar to those of the past. The increase is processor power, however, has lead to marked increase in performance.

As an example of the comparison of computational power between human and machine, in the famous match between Kasparov and Deep Blue [IBM, 1997], Deep Blue was capable of analysing 200 million moves per second. Kasparov, in comparison, was capable of analysing approximately three.

Modern chess computers function by combining the three following main sections into a whole, which is capable of generating and analysing a very large number of moves far faster than a human is able. In order to generate moves, the program will use the board representation to get the positions, search through a "tree" which represents the possible game states, and then evaluate the position using the positional evaluator.

- Board representation and Move Generation

  How is the board represented in computer memory in a way that is efficient and easy to manipulate, and how are the moves derived from and applied to this representation.

- Tree Searching Techniques

  How the program performs a search through the "tree" of possible moves to find the best possible move in a given situation.

- Positional Evaluation

  How the program works out the strategic benefit of a specific position on the board.

This Literature Review is split into sections based on these three issues, and will focus primarily on research based around the solving of these issues in past computer chess programs. There are, of course, a number of other issues that exist when creating a chess computer, but they are not the province of this review.

2 Board Representation and Move Generation

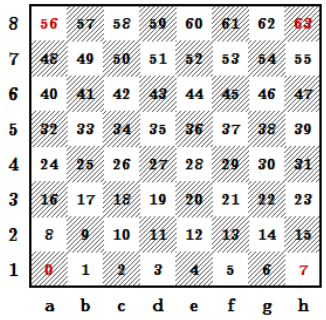
One of the most important aspects of design within a chess computer is how the board is represented in the computer's memory, and how that allows the computer to interpret moves. Conceptually, this is the simplest of the three main sections of the program, and as such is approached first in this review.

When attempting to represent the chess board, it is highly likely that a prospective programmer would first think of storing the 8 x 8 grid as a 2 dimensional array. At first, this seems to be feasible as a useful method, and would certainly work correctly. The issue, however, is not basic functionality, but speed and ease. Whilst the 2D array method could be used, the methods detailed below are the outcome of years of research into the field, and are the solutions that have powered most of the major chess computers.

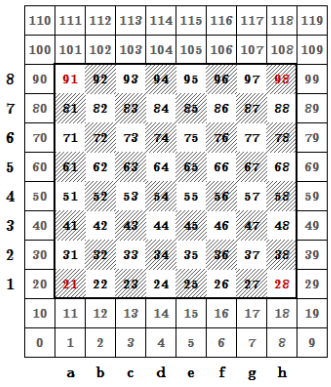
2.1 Offset (or Mailbox) Representation

The Offset (or mailbox) representation, described within *Chess Skill in Man and Machine* [Frey, 1977, p.55-59], works by assigning each of the squares on the board an index in a 64 member array [Shannon, 1950]. The board squares are indexed with a1 being mapped to 0, and h8 being 63, as shown in Figure 1 (a).

More modern versions of the Offset representation, however, use a 10x12 grid instead of the standard 8x8 in order to make it easier to check for illegal moves and the edge of the board. All of the "extra" grid squares which are not part of the chess board are assigned a large number such as 99, in order to designate this area as "out of bounds". This is shown diagrammatically in Figure 1 (b). The use of this larger board is assumed when describing the board below.



(a) Offset representation on a standard chess board.



(b) Offset representation on an expanded 10 x 12 grid.

Figure 1: Diagrams showing indices of squares in the Offset board representation.

To interpret which of the 12 possible distinct chess pieces are on a square, a system of numbering was devised. Traditionally, 0 represents an empty square, 1 represents a pawn, 2 a Knight, 3 a Bishop, 4 a Rook, 5 the Queen, and 6 the King. Positive numbers represent white pieces, and negative numbers the black pieces (e.g. -4 would be a Black Rook).

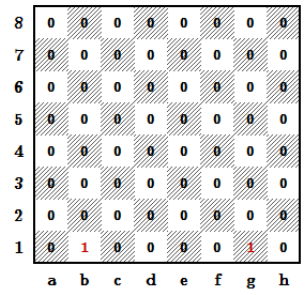
Once the type of piece is determined, it is actually very simple to generate and check legal moves for a specific piece. Taking a Black Knight positioned on e4 for example, we can see that that square is represented by the index 55. From here, the possible positions the Knight could move into are d2, c3, c5, d6, f6, g5, g3 and f2. These squares contain the indices 34, 44, 63, etc. We note the mathematical relationship between the index of the target squares and the current square, and then note these down (e.g. 55 to 34 is -19). Using this set of 8 numbers, we can generate a position for any Knight from any position - the relationships will remain the same no matter the start position.

In order to check if a specific move is legal, the contents of the destination square must be checked. If the square contains a negative integer, then the move is illegal as the square is already occupied by a black piece. If the square contains the number 99, then the move is also illegal as it will be off the board. If, however, the square contains either 0 (empty square) or a positive integer (white piece) then the move is available, and is referred to as *pseudolegal* [Frey, 1977, p.55-59]. This means that the move can be considered to be legal at this stage, though it may prove not to be later in the game logic for some currently unknown reason (e.g. it may leave the King in check). It should be noted that this system will work just as well for the sliding pieces, such as a Bishop, but it will be more complex to calculate.

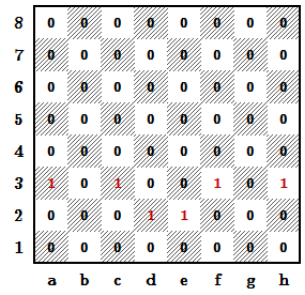
Hyatt [2004] details a further extension of this method, referred to as the 0x88 method, which relies on the use of a hexadecimal numbering system to determine whether or not a proposed move is out of bounds. This representation was created to save storage locations, and was popular upon its release, but over the years it has been almost entirely replaced by the Bit Boards system [Hyatt, 2004].

2.2 Bit Boards (Bitmaps)

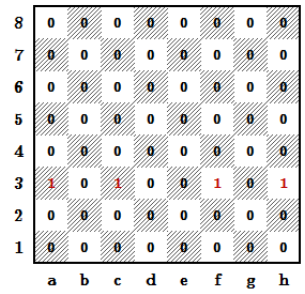
In the late sixties, a number of computer chess groups suggested an alternative to the offset approach [Frey, 1977, Adelson-Velskiy et al., 1970]. The approach, known as the *bit boards* method, takes advantage of the fact that there are 64 squares on a chess board. It does this by using a single 64-bit integer to represent the entirety of the chessboard, with each bit corresponding to a square. A set bit (value of 1) indicates that there is a piece on the square, and an unset (0) bit indicates that the square is empty.



(a) Bit board representing White knight positions at opening



(b) Bit board representing possible moves for the White knights at opening



(c) Bit board representing pseudolegal moves for White knights from opening

Figure 2: Diagrams of three example bit boards for use in White knight move generation

The method relies on the use of many bit boards, each representing a concept such as "All white pawns" or "All pieces attacked by black". As standard, however, there are twelve main bit boards - one to represent each type and colour of piece on the board. Figure 2 (a) below shows the bit board for White knights at the start of the game.

Whilst this approach immediately seems to be rather sensible, it is not until we examine the process of move generation that the true advantage is has over the offset approach it is revealed. By way of example we will calculate the positions that the White knights can move into at the start of the game. First, we retrieve the bit board which represents the positions of the White knights - this is Figure 2 (a). Next, using the positions of the two White knights, the processor would retrieve two bit boards which represent the possible moves of the two knights from the starting squares. In order to work out possible moves for both knights combined, the two bit boards retrieved are logically ORed together to give the bit board shown in Figure 2 (b). The processor then fetches a bit board which represents the positions of all current White pieces. This is then inverted (logical NOT) to give a bit board which shows all squares NOT occupied by white pieces. This resulting board, and the bit board shown in Figure 2 (b) are then logically ANDed together to give the bit board representing all possible *pseudolegal* moves by White knights from the starting position, as shown in Figure 2 (c).

This entire process requires the use of a very small number of instructions when compared to the Offset approach, and this is the most easily demonstrable benefit of bit boards - they are much more computationally efficient [Frey, 1977, p.59]. Whilst the standard approach starts with only 12 bit boards, most chess programs will in fact utilise far more than this [Frey, 1977, p.58]. Due to the modular nature of bit boards, it is simply the work of the programmer creating more bit boards that will transform a very basic system that utilises only the original 12 bit boards into a system which can cope with many more difficult situations and rules.

2.3 Rotated Bit Boards

Developed by A.M. Hyatt and the *Crafty* chess computer team [Hyatt, 1999], Rotated Bit Boards are an evolved version of standard bit boards. They were originally created to increase the efficiency of standard bit boards in order to increase the computational speed of a chess engine. Hyatt noted that working out moves for sliding pieces on standard bit boards takes a lot more work than non-sliding pieces [Frey, 1977, p.55-59], and sought to improve this for the construction of his new chess engine, Crafty. In order to explain the concept of rotated bit boards, we will use an example that will reference the section above on standard bit boards.

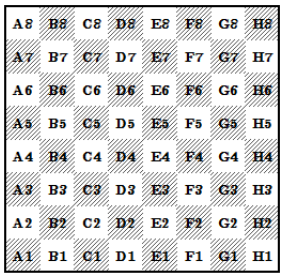
As with standard bit boards, the indices start with the 0th index being the bottom left square, and the 63rd index being the top right. Using that, suppose that a Rook is standing on any specific square within a standard bit board, such as that in Figure 3 (a). It can be seen that moving along the rank one way or another will move through series of indices which are adjacent in memory. When analysing the same rook moving up and down along the file, it can be seen that it will pass through a series of indices which are not adjacent in memory. If adjacent memory locations can be utilised, the efficiency of computation is vastly increased as less "fetch" instructions are needed, and memory access can be simply

pipelined.

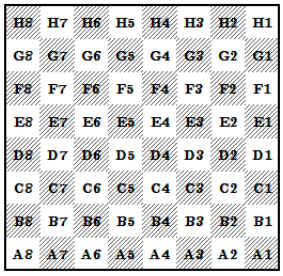
Rotated bit boards are designed to take advantage of this fact by rotating the bit board by a set amount. If we look at the bit board in Figure 3 (b), we can see that the indices are no longer adjacent to one another by rank, but in fact by *file*. This bit board has been rotated by 90 degrees, and therefore the indices in the files are now adjacent to one another. This allows the use of memory pipelining when moving along the file also.

Whilst this solves the problem for Rooks, there still exists the problem of diagonal moves by the Queen and Bishop. To solve this problem, the bit board is rotated by 45 degrees. The outcome of this is shown in Figure 3 (c). In this case, the square marked A1 will be the 0th index, A2 will be the first, B1 will be the second, and so forth. This rotation, therefore, has allowed diagonally adjacent squares to be adjacent in terms of indices. There is a fourth rotated bit board which is used, but it is merely a further 90 degree rotation of Figure 3 (c).

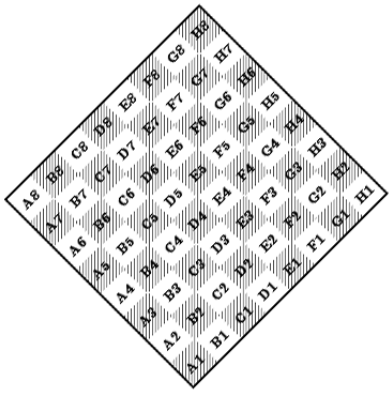
From this point the program is able to utilise two bit boards for representing the movement of Rooks (Figures (a) and (b)) and Bishops (Figure (c) and the fourth bit board which is not shown) and all four of the bit boards for the Queen's movement.



(a) Index positions for a standard chess bit board



(b) Layout of a bit board rotated by 90 degrees



(c) Layout of a bit board rotated by 45 degrees

Figure 3: Diagrams showing the changing indices on rotated bit boards

3 Tree Searching

Of the three main sections in a standard chess program, this is generally the most complex. The phrase "tree searching" refers to the idea that a diagrammatic representation of possible moves in sequence through a game would lead to a structure that looks very similar to a tree [Frey, 1977]. Within the game of chess, the nodes of the tree represent board positions, and the branches of the tree represent possible moves. To give an idea of the complexity of the game and the number of possible positions, it was calculated by Bean [2007] that there will be 69,352,859,712,417 possible nodes at a search depth of 10 ply. This calculation took a week of processing on a 1.9 Ghz Pentium 4.

Tree searching algorithms are responsible for finding and choosing the moves from a specific position, interacting with the positional evaluator to do so. Within the world of computer chess, these algorithms are referred to as either Type A or Type B, as defined by [Shannon, 1950].

As suggested by Frey [1977, p.61], the ideal way to generate the best move for a player in a given situation would be to evaluate every possible move and every outcome from choosing that move. This approach is known as the Shannon Type A strategy - a full width search of all possibilities. Whilst this is fundamentally plausible for a very basic game, it is simply not possible in chess. De Groot [1966] states that there are approximately 35 moves from any given position in chess. Considering that the number of possible moves will increase exponentially as search depth is increased, this is simply far too large a number for the computer to be able to discover the absolutely *best* move every time [Shannon, 1950].

To solve this problem, Shannon [1950] suggested that it would be better to examine a smaller subset of possible moves in a given situation. This is the very definition of the Type B strategy - limiting the search based on some exterior notion of how "good" a move is in a certain situation. There are, according to De Groot [1966], approximately five sensible moves from a specific board position. As such, it can be seen that it is possible to reduce the set of moves that need to be evaluated by the machine by a great deal. This led to the proliferation of the Type B approach, and for many years it was to be the dominant approach. A famous exponent of the Type B search was Chess 3.6, detailed in Levy and Newborn [1982].

More recent chess computers, however, will often use a mixture of the two types of program [Marsland, 1995], performing a Type A search initially, then refining this with a Type B search on moves that prove promising at this point. As such, this review covers research on Type A algorithms initially.

In order to discuss current algorithms, it is pertinent to mention the ancestor on which they are based - the **Minimax** algorithm. Created originally by Von Neumann and Morgenstern [1944], this algorithm is the forerunner of those that are used in modern computer chess. As defined by the Oxford English Dictionary, the Minimax algorithm states that "...for a finite zero-sum game with two players, the smallest maximum loss that a player can choose to risk by a suitable choice of strategy is equal to the greatest minimum guaranteed gain." [OED, 2008]. It does this by assuming that the "best" move for a specific player will be chosen at every node of the game tree; i.e. the player will select the move that represents maximise possible gain on their turn, and that move which minimises the possible detriment on the opponents turn.

This approach, however, means that the computer will have to examine *all* possible branches of the game tree at each node. As stated above, this is entirely impractical, and therefore alternatives to a true minimax search were sought.

Negamax Search The Negamax algorithm is an alteration to minimax that takes advantage of the fact that chess is a zero-sum game to simplify the programming of a minimax search. It does this by using the relation that the minimum possible value for one player is inherently the maximum value for the other player. It is therefore possible to use only maximising operations in the code, by taking the negative of the subtree value that is returned for one of the players [Marsland, 1992].

3.1 The Alpha-Beta Pruning algorithm

The Alpha-Beta Pruning algorithm is the foundation of modern computer chess searches. It was first suggested, though not by name, by Newell et al. [1959]. It was refined over a period of years, most notably by Knuth and Moore [1975]. This algorithm is now the *de facto* standard (as far as there is one) for computer chess tree searching.

In order to explain the method behind Alpha-Beta Pruning, a very basic description of its process is given, followed by a more in-depth look at pseudocode and diagrammatic representations.

Imagine, then, a point in a game of chess between two "players", in which each player is capable of looking ahead by two plies (i.e. they will look ahead until the opponent's next move). It is the turn of White, and the ideal move is being calculated. When White looks forward, they see that they have two possible moves, A and B. They analyse move A, and notice that after Black's next move, the result will be an even position (i.e. no change in perceived strength of that position from the current one). Upon examining move B, however, White notices during analysis that the very first of Black's responses to this move results in the loss of a Knight for White. The Alpha-Beta algorithm, at this point, allows White to stop analysing any more of Black's responses to move B. White can see that there exists *a single* move for Black after move B is played that results in a worse position than *any* of Black's responses if move A is played, and thus move B can be discounted. Whilst there is a possibility that there exists a move after move B that is superior for White, this is immaterial as all possible outcomes from playing move A are immediately superior to possibly losing a Knight. Described in terms of the algorithm, it would be said that analysis of move A gave a *Lower Bound* for the search. Any move with a value lower than this can be safely pruned (i.e. removed from the game tree), as it will not affect the outcome of the algorithm.

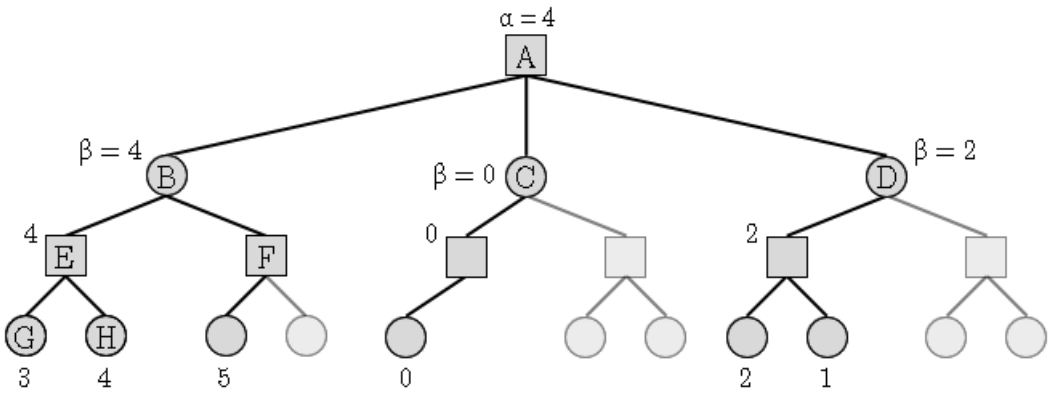


Figure 4: Diagram of a simple game tree, pruned by the Alpha-Beta algorithm.

Figure 4 represents the outcome of a very simple Alpha-Beta search to a depth of 3 plies. The light grey branches represent those that are pruned from the tree during the processing of the algorithm.

As it is rather difficult to understand the function of the algorithm from simply viewing the diagram, it is explained below.

1. Start the process at node A, then descend to the full depth, and take the value $min(G, H)$.
2. Propagate this value, 4, upwards to node E.
3. This value is then assigned to node B as its β value. Node B will therefore be no greater than 4.
4. Check other children of B, and terminate the search if any of the nodes have a value which is greater than the β value of B. F will be β-pruned as its value is at least 5.
5. The β value of B is then assigned to A as its α value. A will therefore be no less than 4.
6. The process is then repeated from step 1, resulting in C having a β value of 0. C will therefore be α-pruned as its value will be 0 or less regardless of the right branch.
7. Finally, the process is repeated for the rightmost branch, acting on node D. D will be α-pruned, as its β value is less than A's α value, and the right branch is therefore irrelevant.
8. This concludes the Alpha-Beta pruning, yielding a final value of 4 for A.

Despite the fantastic savings that Alpha-Beta creates over minimax, it is actually a very simple algorithm, and can be programmed in very few lines of code.

```
1  function alphaBeta(alpha:integer, beta:integer, depthleft:integer):integer;
2     if depthleft == 0 then return evaluate();
3
4     for all moves do
5        begin
6           score := -alphaBeta( -beta, -alpha, depthleft - 1 );
7           if score >= beta then return beta;
8           if score > alpha then alpha := score;
9        end;
10
11    return alpha;
```

Algorithm 1: Pascal-style pseudocode for a Negamax implementation of the Alpha-Beta Pruning algorithm.

It is worth noting that Algorithm 1 is specifically the Negamax version of the Alpha-Beta. This means that the two if statements checking score against *alpha* and *beta* both utilise the same relationship (i.e. greater than, lines 5 and 7). Were this the standard Alpha-Beta, two code segments would be needed to determine the max and min separately, and would utilise both greater than and less than operations. Also worthy of note is the *evaluate*() call on line 2 - this is where the tree-searching algorithm interacts with the program's positional evaluator.

To see the effect that utilising Alpha-Beta over a true minimax search has, we can do some simple calculations. According to Knuth and Moore [1975], the number of terminal positions on the search tree for a minimax search will be d^h, where d represents the number of successors at each node, and h represents the number of plies deep the search is. In comparison, Frey and Atkin [1978] state that the number of terminal positions for Alpha-Beta search is $2d^{h/2}$. We can see from these formulae that the number of nodes within Alpha-Beta is simply twice the square root of the number of nodes in Minimax. As way of example, if we use the values $d = 35$ and $h = 5$, minimax will have approximately 52 million positions, whereas alpha-beta will have approximately 14 thousand. This is an enormous reduction, and one that has made using Alpha-Beta, and indeed the Shannon Type A approach, viable.

3.2 Alpha-Beta Enhancements

Whilst the Alpha-Beta algorithm has met with undoubted success over the years and its use in chess computers is nearly unrivalled, it has maintained this position due to various enhancements. Before analysing any successors or replacements to Alpha-Beta it is important to look at these enhancements, a number of which are compared by Marsland [1983]. Far too many enhancements to Alpha-Beta have been developed to be able to discuss them all within the confines of this report, and as such only the most successful or important are reviewed here. Later in the report, further Alpha-Beta enhancements are described at various points. These are Minimal Window Searching in section 3.3, and Transposition Tables in section 4.4.

Aspiration Search As standard, Alpha-Beta is carried out between the initial bounds of $-\infty$ to ∞. In order to reduce the time taken to process searches, the Aspiration Search technique was developed to limit the "window" which is searched (i.e. between the lower and upper bounds). Marsland [1992] proposes the use of $(MaterialBalance - PawnValue)$ and $(MaterialBalance + PawnValue)$ as the two possible bounds. The gain with this approach is nominally that the search will execute much faster if the minimax value falls within the Aspiration window. Shams and Kaindl [1991], however, point out the problem with this approach - the concept of re-searching. This can be a particular issue as the game approaches a point where a move could elicit a large change in the Material Balance (e.g. near to the end of the game when a checkmate could occur). Iterative Aspiration Search is a slightly more advanced version of the Aspiration search which uses the score returned by the previous iteration of the search as the centre of the search window [Levy and Newborn, 1982].

Whilst Aspiration Searching is an effective tool for increasing the efficiency of Alpha-Beta, it has essentially been replaced by Minimal-Window search, as described by Marsland [1983]. This is described in more detail in section 3.3 - The Negascout Algorithm.

Quiescence Search As far back as Shannon [1950], there has been an understanding that positions should only be evaluated if they are "relatively quiescent". A quiescent position in chess is defined as one which will lead very quickly to a dead position. These positions are chosen as they are able to be searched easily without the need for further searching. The main idea behind the Quiescence Search is to enable the processing of a more accurate positional evaluation, but it is also designed to limit the Horizon Effect, discussed in section 4.3 [Marsland, 1992].

Progressive/Iterative Deepening Progressive Deepening is a type of Move Ordering algorithm. It attempts to "sort" the list of possible moves so that the *best* move in that situation is searched first.

In order to discover the concept of Progressive Deepening, De Groot [1966] studied human players playing chess, and noticed their propensity to re-examine similar branches of a game tree multiple times [Gobet and Charness, 2006]. This led to the idea of the Progressive, or Iterative, deepening search - where a series of searches will be conducted through the tree to successively greater depths. As more iterations are conducted, the "confidence" that the best move will be searched first is increased [Schaeffer, 1989a]. It therefore has great use in time-limited situations, as the deepening can be cut-off when needed and the best move at that point returned.

The Killer and History Heuristics The Killer Heuristic relies on the notion of one move *refuting* another. If a move at level $n-1$ in an n-ply tree is cut off from the tree by a specific move at level n, the move at $n-1$ is said to be refuted by the move at n. The Killer Heuristic is based on this simple premise, first devised by Greenblatt et al. [1967].

In terms of a chess computer program, this heuristic will often be used to store a list of the moves at each search depth that have been shown previously to refute the most other moves at that depth. These moves, known as *killer* moves, can then be retrieved if a similar position is reached again in that game. According to Marsland [1992], the consistent use of the Killer Heuristic early in searches increases the likelihood of an early cut-off to a tree search, thus increasing the efficiency of the search.

The History Heuristic [Schaeffer, 1989b] is a powerful move ordering method which takes some inspiration from the Killer Heuristic. The History Heuristic utilises a number of tables to keep track of every legal move in a search tree, and notes how successfully that move refutes others. In this case, it treats a particularly meritorious (in terms of value) move as a refutation as well as one that actually causes a cut-off in the game tree. It then stores this list, and it is ordered so that the most historically successful moves are attempted first. Similarly to the Killer Heuristic, it operates on the principle that if a move has proved successful in the past, it is likely to be if used in the same way again. One of the keys to the success of the History heuristic is that it is accessible from anywhere within the game tree [Marsland, 1992].

3.3 The Negascout and MTD(f) Algorithms

Negascout and MTD(f) are both alternatives to Alpha-Beta that have been introduced more recently than the standard Alpha-Beta search. To understand their operation, a procedure known as Minimal Window Searching must be introduced.

Minimal Window Searching builds on Aspiration Window Searching but attempts to limit the size of the search window to the smallest value possible. It does this by creating an aspiration window based on

the current minimax value, in a form such as $-value, value$. It will typically then use iterative deepening to get new minimax values to assign to the minimal window bounds.

In order to increase the efficiency of Alpha-Beta, a procedure named Test was developed by Pearl [1980]. This type of procedure is known as a proof procedure. A proof procedure makes an assertion about a specific search, and then determines a binary true/false outcome. Within tree searching, Test was used to determine whether a minimax value of a subtree was above or below the value expected. Test is embedded within an algorithm called Scout, and comparisons between this and Alpha-Beta have yielded results to suggest that Scout will in fact be more efficient [Plaat et al., 1994].

The Negascout Algorithm [Reinefeld, 1989] is a variant of the Scout algorithm which uses a Negamax framework, and produces even better results in comparison to Alpha-Beta [Reinefeld, 1989].

According to Plaat et al. [1994], however, the advantages of Negascout over Alpha-Beta are only clearly apparent when using *standard* Alpha-Beta. When using some of the enhancements suggested earlier in this section, the advantage of Negascout is greatly reduced, in proportion to how competent the move ordering of the Alpha-Beta is.

Plaat et al. [1994] suggest a new algorithm which builds on the Test procedure used in the Scout-variant algorithms such as Negascout. The algorithm, known at this stage as the MT algorithm, does this by enhancing Test with the use of memory. MT, in fact, stands for Memory enhanced Test. Very similarly to the History Heuristic, the MT algorithm will check to see if the value of a node is already stored within memory before that node is searched.

The MT algorithm, however, will merely return a bound on the minimax value, and in order to determine the actual minimax value a slight enhancement must be made. The MTD(f) algorithm, which is a *driver* for MT, has a value passed to it which determines where it should start searching for the minimax value. By the use of Iterative Deepening, this value will be the value of the previous iteration, which means that the MTD(f) algorithm will increase in efficiency as the accuracy of the initial value for minimax is increased.

The pseudocode below represents the operation of the MTD(f) algorithm, though it is important to note that this algorithm will ordinarily be called from within an iterative deepening framework.

```
1  function MTDF(root : node_type; f : integer; d : integer) : integer;
2        g := f;
3        upperbound := +INFINITY;
4        lowerbound := -INFINITY;
5        repeat
6              if g == lowerbound then beta := g + 1 else beta := g;
7              g := AlphaBetaWithMemory(root, beta - 1, beta, d);
8              if g < beta then upperbound := g else lowerbound := g;
9        until lowerbound >= upperbound;
10       return g;
```

Algorithm 2: Pascal-style pseudocode for the MTD(f) Algorithm [Plaat et al., 1994].

4 Positional Evaluation

Marsland [1992] states that this is the most important part of any computer chess program. It provides the logic that allows the program to determine how *valuable* a specific piece or position is, and as such is the backbone of the program. It is, however, the area which is the least effectively understood in terms of computer chess as it is reliant on relatively ambiguous ideas such as "pawn structure" at a specific point in a game.

4.1 Basic Positional Evaluation - Material Balance

Material Balance is by far the simplest concept when evaluating certain positions on a chess board, and is only the basis of the Positional Evaluator in all but the very simplest systems [Marsland, 1992]. A player's material is simply the pieces which that player has left in play on the chess board at a given time. When assessing the material balance, therefore, it is a very simple process of looking at which pieces each player has left, determining the value of each piece, and comparing the overall value of each side's pieces.

Values of individual pieces are often standardised, though it is really rather arbitrary what value they are given as long as the ratios between them are similar. Piece value is often defined in terms of number of pawns, e.g. this piece is worth n pawns. The standard chess piece values are defined below.

$Pawn : 1$ $Bishop/Knight : 3$ $Rook : 5$ $Queen : 9$ [Frey, 1977]

The King is, however, slightly more problematic. As a checkmate will end the game, it is difficult to assign a value to it. A lot of chess programs will therefore assign an artificially large value to the King (e.g. 1000) in order to ensure that search nodes near Checkmate positions are more valuable than others. During the endgame, however, the value of the King will be evaluated to approximately 4 if it is not in danger of attack.

Material Balance is usually the most important aspect of a Positional Evaluator, and the evaluation of a position will often be weighted towards the Material Balance over other measures [Marsland, 1992].

4.2 Strategic Balance

From this report, it can be noticed that the level of chess knowledge required to understand most of the concepts within a chess program is actually rather lower than expected. It is possible to understand the tree searching algorithms and even the board representations without a knowledge deeper than how the pieces move. Strategic Balance, however, is the point at which the chess skill of the programmer can prove important.

Strategic Balance is a measure of relatively complex concepts such as king safety, pawn structure, and square control. The difference between this section and others within this report is that it is very possible to change a poor chess program into an excellent one by simply creating more rules to define the strategic balance. If the only idea a chess program has of strategy is that it must eventually checkmate the opponent, it will be challenging for it to defeat high-level human players.

As this section deals exclusively with concepts of chess, rather than chess computers, no more will be said about the idea of Strategic Balance. If more information is desired about these topics, then please consult Chess.com [2009].

4.3 The Horizon Effect

The Horizon Effect is an issue which even modern computer chess programs have problems with. It is the concept of being unable to predict a move that is beyond the search "horizon" of the program (i.e. maximum search depth), which means that it will be unable to cope with the move correctly. This effect is exacerbated if the program is not making use of an Acquiescence Search, of if it is using a poor one. Figure 5 demonstrates the Horizon Effect in practice. In this position, if a chess program suffered badly from the Horizon Effect, it would assume that moving the Black Pawns down a rank to protect the Queen would be an ideal move. In practice, however, this would likely lead to the White player capturing the black Queen by moving the Bishop diagonally down the line, taking the pawns in turn [Marsland, 1992].

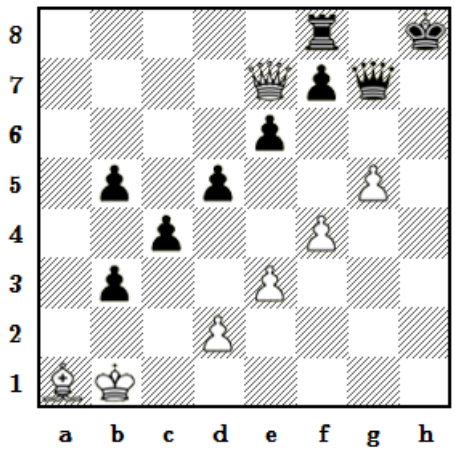


Figure 5: Diagram showing an example of the Horizon Effect. (Black to move)

4.4 Endgame and Transposition Tables

For chess computers, the endgame represents the greatest challenge of any phase of play. Endgames are often 20 moves or more in an exact order [Frey, 1977], which is a vastly deeper search than almost all chess computers are capable of dealing with. To counteract this problem, there are a few methods that have proved invaluable after testing with numerous implementations.

Transposition Tables (see Marsland [1992]) Transposition tables are databases of stored moves that have been searched previously, and are used to reduce the amount of searching needed without seriously affecting the outcome of searches. They do this by storing a number of different pieces of information, such as Alpha-Beta value.

Transposition tables allow a number of different improvements to the search procedure. Firstly, they will act in a similar fashion to the Killer Heuristic, in that moves from the table that are shown to cut-off the search a number of times will plausibly allow searches to be cut off again at this point.

Secondly, when used with a minimal window search and iterative deepening, the use of the Alpha-Beta value stored in a transposition table node will often allow the search procedure to be optimised by providing a value to start the search at.

Most importantly, however, is the fact that it will allow the program to "recognise" board positions and potential moves from these positions by examining the corresponding transpositions.

Transposition Tables are one of the most important part of endgame play, as they allow move sequences to be easily stored and retrieved without so much emphasis on full-depth tree searching.

Positional Evaluation Changes In order to help deal with the problems of the endgame, the positional evaluator can be changed to adjust values of pieces and how they are related. Strategic Balance will be updated to give more weighting to important endgame aspects such as "passed pawns" or position of the opponent's King. As stated in the Strategic Balance section, it is rather complex to detail the particulars of *how* the game balance changes throughout the game, but the most effective chess programs will utilise a large number of different evaluators.

The combination of these factors has enabled modern chess computers to increase in their ability to succeed in playing during the endgame, but it is still often the weakest part of a chess computer's playing [Marsland, 1992].

5 Conclusion

At the beginning of this paper, the three main sections in a computer chess program were listed. In the first section, board representations were looked at, as were the differences between using a basic array-type approach and the bit board representation. Bit board representations were found to be more efficient and more flexible, but the learning curve required to easily manipulate bitmaps is rather steep.

Following on in the second section, the history of tree searching algorithms, beginning with the Minimax algorithm, was looked at. Its replacement, the Alpha-Beta pruning algorithm, was then reviewed. The Alpha-Beta algorithm has been the standard algorithm implemented into chess computers for thirty years, and the enhancements listed later in the second section serve to show how it can be updated to (almost) keep up with modern algorithms. Possible replacements for Alpha-Beta were briefly discussed, and it was noted that the MTD(f) algorithm is the fastest algorithm yet created for computer chess searching.

The most important part of a chess program - the positional evaluator, was looked at in the third and final section. The problems that can arise for a novice chess player attempting to program a chess computer were seen after looking at Strategic Balance, but also it was discovered that the highest impact part of the positional evaluator is the far-simpler Material Balance.

Chess computers have been shown to be an interesting mix of complex concepts and very simple, well-produced algorithms that have been honed over six decades of research. With new evolutions being created all the time, the field is being constantly updated still to this day. As the problem ages and becomes better-known, people are trying more and more diverse tactics to succeed at computer chess. From the use of Combinatorial Game Theory in Elkies [1996], to Gro et al. [2002] using genetic programming to attempt to solve endgame problems, there are increasingly varied methods being attempted.

Dvorsky [2006] discusses his thoughts on the future of computer chess on the Institute for Ethics and Emerging Technologies website. He seems reasonably sure that the time has passed where a human can provide adequate challenge for a machine - modern machines such as Rybka are making moves that are confusing even the Grandmasters. In the years to come, increased computational power will edge computers closer and closer to finally solving the game.

Regardless of this, computer chess is a topic that will continue to entertain human minds for many years to come. The problem which fascinated Shannon, Turing and others in the late forties is as interesting today as it was at the end of the Second World War.

References

Oxford English Dictionary. http://dictionary.oed.com/cgi/entry/00310238, 2008.

G.M. Adelson-Velskiy, V.L. Arlazarov, A.R. Bitman, A.A. Zhivotovsky, and A.V. Uskov. Programming a Computer to Play Chess. *Russian Mathematical Surveys*, 25:221–262, 1970.

R. Bean. Counting Nodes in Chess, 2007.

Chess.com. How to Play Chess: Rules & Basics. http://www.chess.com/learn-how-to-play-chess.html, 2009.

A.D. De Groot. *Thought and Choice in Chess*. Mouton De Gruyter, 1966.

George Dvorsky. The Future of Chess. http://ieet.org/index.php/IEET/more/dvorsky20061208/, 2006.

N. D. Elkies. On Numbers and Endgames: Combinatorial Game Theory in Chess Endgames. In *In Games of No Chance*, pages 135–150. Cambridge University Press, 1996.

P.W. Frey. *Chess Skill in Man and Machine*. Springer-Verlag, 1977.

P.W. Frey and L.R. Atkin. Creating a Chess Player. *Byte Magazine*, 1978.

F. Gobet and N. Charness. Expertise in Chess. In *Chess and games. Cambridge handbook on expertise and expert performance*, pages 523–538. Cambridge University Press, 2006.

R.D. Greenblatt, D.E. Eastlake, and S.D. Crocker. The Greenblatt Chess Program. In *Procs. Fall J. Computer Conf.*, volume 31, pages 801–810. Thompson Books, 1967.

R. Gro, K. Albrecht, W. Kantschik, and W. Banzhaf. Evolving chess playing programs, 2002.

R.M. Hyatt. Rotated Bitmaps, a New Twist on an Old Idea. *International Computer Chess Association Journal*, 22(4):213–222, 1999. http://www.cis.uab.edu/hyatt/bitmaps.html.

R.M. Hyatt. Chess Program Board Representations. http://www.cis.uab.edu/hyatt/boardrep.html, 2004.

IBM. Kasparov vs. Deep Blue - The Rematch. http://www.research.ibm.com/deepblue/, 1997.

D.E. Knuth and R.W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6:293–326, 1975.

D. Levy and M. Newborn. *All About Chess and Computers*. Springer-Verlag, 1982.

T.A. Marsland. Relative Efficiency of Alpha-Beta Implementations. In *Procs. 8th Int. Joint Conf. on Art. Intell*, pages 763–766. Kaufmann, 1983.

T.A. Marsland. *Encyclopedia of Artificial Intelligence* , chapter Computer Chess and Search, pages 224–241. Wiley, New York, 1992.

T.A. Marsland. The Anatomy of Chess Programs. http://www.cs.ualberta.ca/~tony/ICCA/anatomy.html, 1995.

A. Newell, J.C. Shaw, and H.A. Simon. Chess Playing Programs and the Problem of Complexity. *IBM Journal*, 2(4):320, 1959.

J. Pearl. Asymptotical Properties of Minimax Trees and Game Searching Properties, 1980.

A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. A New Paradigm for Minimax Search. Discussion paper, Department of Computing Science, University of Alberta, Edmonton, 1994.

A. Reinefeld. *Spielbaum - Suchverfahren*. Springer, 1989.

J. Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11:1203–1212, 1989a.

J. Schaeffer. Distributed Game Tree Searching. *Journal of Parallel and Distributed Computing*, 6(2): 90–114, 1989b.

R. Shams and H. Kaindl. Using Aspiration Windows for Minimax Algorithms. In *Procs. 8th Int. Joint Conf. on Art. Intell*, pages 192–197. Kaufmann, 1991.

C.E. Shannon. Programming a Computer for Playing Chess. *Philosophical Magazine*, 41(314), 1950.

J. Von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.